**I) Слоеве на OSI модела:** Physical, Data link, Network, Transport, Session, Presentation and Application.

**Слоеве на Интернет модела:** Physical, Data link, Network, Transport, Application.

**II) Комуникации и мрежи:** Мрежово (IP) ниво: Нивото отговаря за преноса на пакети от възела-източник до възела-цел в същата, а взможно и в различни, свързани помежду си мрежи.За целта на всеки възел се присвоява уникален адрес, който е 32 битово число.а.b.c.d,където 0≤ а,b,c,d≤255.

Транспортно ниво: Транспортното ниво отговаря за преобразуването на съобщението в пакети от данни. То поддържа два основни протокола:-TCP :канално ориентиран;-UDP: дейтаграмно ориентиран.

TCP протокол – създава двупосочен канал между двете приложни нива.От съобщението се формират пакети от данни, като се добавят номерата на изходния **порт** и на **порта** цел.Възстановява от пакетите цялостното съобщение като коригира евентуалните грешки.

UDP протокол: Дейтаграмно ориентиран.Приложното ниво отговаря за сглобяването на съобщението от дейтаграмите.Няма корекция на грешките. Висока производителност.

Модел Клиент-Сървър: Сърверът предлага дадена услуга, която се идентифицира еднозначно с IP адреса на сървера и номер на порт, който е число между 0 и 65535 (0-1024 – системни портове) Клиентът се свързва към вече обявена услуга. След свързването диалогът се определя от протокола на услугата.

**III) Програмиране на приложно ниво I част:** Взаймодействие между клиент и сървър: *Сървър*: 1) Инициализация и наименоване на сокет на услугата (**socket, bind, listen**) 2) Заспива, чакайки клиент При контакт от клиента, създава сокет на връзката (**аccept**)3) Обработва заявката и изпраща отговор и т.н. (**read, write**) 4) Затваря връзката (**close**)**;** *Клиент*: 1) Старт и инициализация на сокет на връзката (**socket**) 2) Активиран, изпраща заявка към сървера (**connect**) 3) Обработва отговора, изпраща евентуално нова заявка и т.н. (**read, write**)4) Затваря връзката (**close**)

Адресиране на socket: Структурите за socket:

struct sockaddr\_in **{**

short int sin\_family; /\*Protocol family\*/

unsigned short int sin\_port; /\*Port number\*/

struct in\_addr sin\_addr; /\*Internet address\*/ **};**

struct in\_addr **{** unsigned long int s\_addr; **}**

Създаване на socket услугата: int socket(int domain, int type, int protocol)**;** **domain** може да е **AF\_UNIX, AF\_INET, AF\_ISO; type** е **SOCK\_STREAM, SOCK\_DGRAM;** **protocol -> 0;** Връща дескриптор на сокета.

Наименоване на услугата: int bind(int sockfd, struct sockaddr \*my\_addr, int addrlen)**;** **sockfd** е дескриптор на socket-a (стойността върната от socket())**;** **sockaddr** e указател към структура, наименоваща socket**;** **addrlen** e дължината на горната структура.

Деф. на опашка: int listen(int sockfd, int backlog); **backlog** e дълж. на опашката със заявки, чакащи за сървера.

**III) cont’d):** Създаване на socket връзката: int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen); **sockfd** е дескриптор на **socket-a** на услугата**;** **аddr** е адрес на структура, в която **accept** записва данните на клиента, който се обръща за услугата. **Аddrlen** е указател към цяла стойност, задаваща дължината на горната структура**.** Връщадескриптор на socket-a на връзката.

ЧЕТЕНЕ/ПИСАНЕ В КОМУНИКАЦИОННИЯ КАНАЛ

**ssize\_t read(int sockfd, void \*buf, size\_t count);**

**ssize\_t write(int sockfd, void \*buf, size\_t count);**

ЗАТВАРЯНЕ НА КАНАЛА

**int close(int sockfd); в unistd.h**

(Пример:)ПРОСТ СЪРВЕР

int main(){

int server\_sockfd, client\_sockfd;

int server\_len, client\_len;

struct sockaddr\_in server\_address;

struct sockaddr\_in client\_address;

server\_sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

server\_address.sin\_family = AF\_INET;

server\_address.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

server\_address.sin\_port = 9734;

server\_len = sizeof(server\_address);

bind(server\_sockfd, (struct sockaddr \*)&server\_address, server\_len);

listen(server\_sockfd, 5);

while(1) {char ch;

printf("server waiting\n");

client\_len = sizeof(client\_address);

client\_sockfd = accept(server\_sockfd, (struct sockaddr \*)&client\_address, &client\_len);

while( read(client\_sockfd, &ch, 1) !=0)

printf (“server receives = %c\n”,ch);

printf(“server closes\n”);

close(client\_sockfd);}}

**III cont’d)** СВЪРЗВАНЕ НА КЛИЕНТ КЪМ СЪРВЕР

#include <sys/types.h>

#include <sys/socket.h>

connect(client\_sockfd, (struct sockaddr

\*)&client\_address, client\_len);

ПРОСТ КЛИЕНТ:

int main(){

int sockfd;int client\_len;int i;

struct sockaddr\_in client\_address;

int result;char ch = 'A';

**/\* Create a socket for the client. \*/**

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

**/\* Name the socket, as agreed with the server. \*/**

client\_address.sin\_family = AF\_INET;

client\_address.sin\_addr.s\_addr =

inet\_addr("127.0.0.1");

client\_address.sin\_port = 9734;

client\_len = sizeof(client\_address);

**/\* Now connect our socket to the server's socket.\*/**

result = connect(sockfd, (struct sockaddr \*)&client\_address, client\_len);

if(result == -1) {perror("oops: clienta");

exit(1);}

**/\* We can now read/write via sockfd. \*/**

for (i=0; i<=9;i++) {

write(sockfd, &ch,1);

printf("client sends= %c\n", ch);

ch++;} close(sockfd);exit(0);}

СТАНДАРТНО ПРЕДСТАВЯНЕ

**#include <netinet/in.h>**

**unsigned long int htonl(unsigned long int host)**

**unsigned short int htons(unsigned short int host)**

**unsigned long int ntohl(unsigned long int net)**

**unsigned short int ntohs(unsignet short int net)**

Пример:**server\_address.sin\_port = htons(9734);**

**IV) Програмиране на Приложно ниво II част:**

МРЕЖОВА ИНФОРМАЦИЯ:

#include <netdb.h>

**struct hostent \*gethostbyaddr(const void \*addr, size\_t**

**len, int type);**

\*addr ->pointer to struct in\_addr

len -> sizeof(struct in\_addr),

type -> AF\_INET

**struct hostent \*gethostbyname(const char \*name)**

struct hostent{

char \*h\_name; /\*host name\*/

char \*\*h\_aliases; /\*list of aliases\*/

int h\_addrtype; /\*address type (AF\_INET)\*/

int h\_length; /\*address length (4)\*/

char \*\*h\_addr\_list /\*array of struct in\_addr \*s \*/};

**ПОДОБРЕН КЛИЕНТ:** Следващият код показва, как сърверът може да се специфицира с име:

#include <sys/types.h, sys/socket.h, stdio.h, netinet/in.h, arpa/inet.h, unistd.h, netdb.h>

int main(int argc, char \*argv[] ) **{**

int sockfd;int client\_len;int i;

char\* host;struct sockaddr\_in client\_address;

struct hostent \*hostinfo;int result;

char ch = 'A';

if(argc == 1) host = "localhost";

else host = argv[1];

hostinfo = gethostbyname(host);

if(!hostinfo) **{** fprintf(stderr, "no host: %s\n", host); exit(1); **}**

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

client\_address.sin\_family = AF\_INET;

client\_address.sin\_addr = \*(struct in\_addr\*)\*hostinfo->h\_addr\_list;

client\_address.sin\_port =htons(9734);

client\_len = sizeof(client\_address);

**IV cont’d 1)** КОЙ СЪМ АЗ?: **int gethostname (char\* name, int length)**;

ПРЕОБРАЗУВАНЕ НА IP АДРЕСА КЪМ СТАНДАРТНА “DOT” ФОРМА:

 **#include <arpa/inet.h>**

**char \*inet\_ntoa(struct in\_addr addr)**;

ПРИМЕР – ИНФОРМАЦИЯ ЗА СЪРВЕРА

int main(int argc, char \*argv[]){

char **\*host**, **\*\*names**, **\*\*addrs**;

struct hostent **\*hostinfo**;

**/\* Set the host in question to the argument supplied with the getname call, or default to the user's machine. \*/**

if(argc == 1) {char myname[256];

**gethostname(myname, 255);**

host = myname;}else host = argv[1];

**/\* Make the call to gethostbyname and report an error if no information is found. \*/**

hostinfo = gethostbyname(host);

if(!hostinfo) {fprintf(stderr, "cannot get info for host: %s\n", host);

exit(1);}

**/\* Display the hostname and any aliases it may have. \*/**

printf("results for host %s:\n", host);

printf("Name: %s\n", hostinfo -> h\_name);

printf("Aliases:");

names = hostinfo -> h\_aliases;

while(\*names) {printf(" %s", \*names);

names++;}printf("\n");

**/\* Warn and exit if the host in question isn't an IP host. \*/**

if(hostinfo -> h\_addrtype != AF\_INET) {

fprintf(stderr, "not an IP host!\n");

exit(1);}

**/\* Otherwise, display the IP address(es). \*/**

addrs = hostinfo -> h\_addr\_list;

while(\*addrs)

{printf(" %s", inet\_ntoa(\*(struct in\_addr \*)\*addrs));addrs++;}

printf("\n");exit(0);}

**IV cont’d 2) СЪРВЕР ЗА МНОГО КЛИЕНТИ**

int main(){

int server\_sockfd, client\_sockfd;

int server\_len, client\_len;

int server\_n = 0;

struct sockaddr\_in server\_address;

struct sockaddr\_in client\_address;

server\_sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

server\_address.sin\_family = AF\_INET;

server\_address.sin\_addr.s\_addr =

htonl(INADDR\_ANY);

server\_address.sin\_port = 9734;

server\_len = sizeof(server\_address);

bind(server\_sockfd, (struct sockaddr \*)&server\_address, server\_len);

**/\* Create a connection queue, ignore child exit details and wait for clients. \*/**

listen(server\_sockfd, 5);

signal(SIGCHLD, SIG\_IGN);

while(1) {char ch;printf("server waiting\n");

**/\* Accept connection. \*/**

client\_len = sizeof(client\_address);

client\_sockfd = accept(server\_sockfd, (struct sockaddr \*)&client\_address,&client\_len);

**/\* Fork to create a process for this client\*/**

server\_n++;

if(fork() == 0) {

**/\* If we're the child, we can now read/write to the client on client\_sockfd. The five second delay is a demonstration. \*/**

while (read(client\_sockfd, &ch, 1)!=0) {

printf("server %d receives=%c\n",server\_n,ch)sleep(5);}

printf ("server closes\n");

close(client\_sockfd);

exit(0);}

else {close(client\_sockfd);}}}

**IV cont’d 3) I/O МУЛТИПЛЕКСИРАНЕ**

МНОЖЕСТВО ОТ ДЕСКРИПТОРИ

#include <sys/time.h>

#include <sys/types.h>

#include <unistd.h>

**FD\_SET(int fd, fd\_set \*set); Add fd to the set.**

**FD\_CLR(int fd, fd\_set \*set); Remove fd from the set.**

**FD\_ISSET(int fd, fd\_set \*set); Return true if fd is in the set.**

**FD\_ZERO(fd\_set \*set); Clear the set.**

ФУНКЦИЯ SELECT

**int select(int numfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);**

**numfds –** проверяват се дескриптори в интервала [**0, numfds-1**] ;

**readfds , writefds, exceptfds** - множествата се проверяват, за готовност за четене, за писане и за изключение;

**struct timeval { // timeout**

**int tv\_sec; // seconds**

**int tv\_usec; // microseconds };**

След изпълнение select връща:-1 - при грешка;

0 - при timeout;-общ брой на дескритори, при които има

готовност за дадената операция. Освен това в множествата, сочени от **readfds, writefds, exceptfds** остават само дескрипторите на файлове с готовността за дадената операция.

СХЕМА НА СЪРВЕР, ОБСЛУЖВАЩ МНОГО

КЛИЕНТИ, ОТ КОИТО ТОЙ САМО ЧЕТЕ

множество дескриптори setd, setw;

Създаване на socket на услугата – sockserv;

setw = {sockserv };

while (1) {setd = setw;

select (..,setd,NULL,NULL,NULL);

for i in setd {

if (i == sockserv) {

sockconn = accept(..);

setw = setw + sockconn; }

else обслужва се i-тия клиент (ako i-тия клиент изпраща низ с нулева дължина, неговият дескриптор се премахва от setw);}}

**V) Java Socket-и.** Java интерфейси: **Пример:**

Socket MyClient;

try **{** // Създаване на клиентски сокет

MyClient = new Socket("Machine name", PortNumber); **}**

catch (IOException e) **{** System.out.println(e); **}**

// Създаване на I/O потоци за клиента

DataInputStream input;

PrintStream output;

try **{**

input = new

DataInputStream(MyClient.getInputStream());

output = new

PrintStream(MyClient.getOutputStream()); **}**

catch (IOException e) **{** System.out.println(e); **}**

int read(byte[] b) – чете байтове и записва в байт масив, като прочита най-много b.length байта

int read(byte[] b, int off, int len) – чете най-много len байта

void write(byte[] buf, int off, int len) – пише len байта от масива от байтове buf в изходния поток

void write(int b) – записва байта в изходния поток

Затваряне на връзката: **Пример:**

try **{**

output.close();

input.close();

MyClient.close(); **}**

catch (IOException e) **{** System.out.println(e); **}**

Създаване на сървърен сокет: **Пример:**

ServerSocket MyService;

try **{** MyServerice = new ServerSocket(PortNumber); **}**

catch (IOException e) **{** System.out.println(e); **}**

Socket clientSocket = null;

try **{** clientSocket = MyService.accept(); **}**

catch (IOException e) **{** System.out.println(e); **}**

**V cont’d 1)** Създаване на I/O потоци за сървъра:

**Пример:**

DataInputStream input;

PrintStream output;

try **{**

input = new

DataInputStream(clientSocket.getInputStream());

output = new

PrintStream(clientSocket.getOutputStream()); **}**

catch (IOException e) **{** System.out.println(e); **}**

WINSOCK интерфейс: Аналогичен на стандартния socket интерфейс: SOCKET WSAAPI socket(\_\_in int af, \_\_in int type, \_\_in int protocol)**;** int bind(\_\_in SOCKET *s*, \_\_in const struct sockaddr \**name*, \_\_in int *namelen* )**;** int listen(\_\_in SOCKET *s*, \_\_in int *backlog* )**;** SOCKET accept(\_\_in SOCKET *s*, \_\_out struct sockaddr \**addr*, \_\_inout int \**addrlen*)**;** int connect(\_\_in SOCKET *s*, \_\_in const struct sockaddr \**name*, \_\_in int *namelen*)**;**

UDP протокол: Дейтаграмен протокол, който е несигурен, т.е. не се гарантира получаването на изпратените дейтаграми**;** Гарантира се само целостта на дейтаграмата (ако се плучи)**;** Не гарантира, че реда на получаване е същият на изпращане**;** С по висока производителност от TCP.UDP socket: int socket(int family, int type, int proto); **Пример:**

int mysock;

struct sockaddr\_in myaddr;

mysock = socket(AF\_INET, SOCK\_DGRAM, 0);

if (sock < 0) **{** /\* ERROR \*/ **}**

myaddr.sin\_family = AF\_INET;

myaddr.sin\_port = htons( 1234 );

myaddr.sin\_addr = htonl(INADDR\_ANY);

bind(mysock, &myaddr, sizeof(myaddr));

**V cont’d 2)** Пращане на UDP дейтаграми:

ssize\_t sendto(int sockfd, void \*buff, size\_t nbytes,int flags, const struct sockaddr\* to, socklen\_t addrlen);

**sockfd** е UDP socket, **buff** е адреса на данните (с дълж. **nbytes**), **to** е адреса на sockaddr съдържащ адреса на получателя. Връща броя на пратените байтове (може да не са получени) или -1.

Получаване на UDP дейтаграми:

ssize\_t recvfrom(int sockfd, void \*buff, size\_t nbytes, int flags, struct sockaddr\* from, socklen\_t \*fromaddrlen);

**from** е адреса на sockaddr на пращача. Връща броя на получените байтове или -1. Ако буфера не е достатъчно, данните се губят завинаги. Можем да получим 0 байта. И при двете няма код за грешка ако данните не са получени.

UDP echo server:

// Предния пример за UDP

while(1) **{**

len = sizeof(cliaddr);

msglen = recvfrom(mysock, msgbuf, MAXLEN, 0, cliaddr, &clilen);

sendto(mysock, msgbuf, msglen, 0, cliaddr, clilen); **}**

Връзка при UDP: UDP сокета може да извика connect() Това само указва на ОС адреса на получателя. Не се извършва handshake за съществуване. Не се пращат никакви данни по мрежата при connect на UDP сокет Когато се „свърже“ може да се ползват функциите за четене и писане (даже с null dest адрес). Само дейтаграми от другия peer ще бъдат върнати.

**VI)** **Remote Procedute Call (RPC).** Участници: **Caller**: a program which calls a subroutine**; Callee**: a subroutine or procedure which is called by the caller**;** **Client**: a program which requests a connection to and service from a network server**;** **Server**: a program which accepts connections fromand provides services to a client. Тhe caller always executes as a client process, and the callee always executes as a server process.

RPC Механизъм: (Remote Procedure Call) 1) The caller program must prepare any input parameters to be passed to the RPC. Note that the caller and the callee may be running completely different hardware. 2) The calling program must somehow pass its data to the remote host which will execute the RPC. The RPC receives and operates on any input parameters and passes the result back to the caller. 3) The calling program receives the RPC result and continues execution.

Проблеми на представянето: The solution to this problem involves the adoption of a standard for data interchange. One such standard is the ONC external data representation (XDR). XDR is essentially a collection of C functions and macros that enable conversion from machine specific data representations to the corresponding standard representations and vice versa. It contains primitives for simple data types such as int, float and string and provides the capability to define and transport more complex ones such as records, arrays of arbitrary element type and pointer bound structures such as linked lists. In computer science, a **RPC** is an [inter-process communication](http://en.wikipedia.org/wiki/Inter-process_communication) that allows a [computer program](http://en.wikipedia.org/wiki/Computer_program) to cause a [subroutine](http://en.wikipedia.org/wiki/Subroutine) or procedure to execute in another [address space](http://en.wikipedia.org/wiki/Address_space) (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction.

**VI cont’d)** RPC Call Binding: An RPC application is formally packaged into a *program* with one or more *procedure* calls. The RPC program is assigned an integer identifier known to the programs which will call its procedures. Each procedure is also assigned a number that is also known by its caller. RPC uses a program called **portmap** to allocate port numbers for RPC programs. When an RPC **program** is started, it registers itself with the portmap process running on the same host. The portmap process then assigns the TCP and/or UDP port numbers to be used by that application. Prior to calling the remote procedure, the *caller* also contacts portmap in order to obtain the corresponding port number being used by the application whose procedures it needs to call. The correct procedure is reached through the use of a dispatch table in the RPC program. The same registration process that establishes the port number also creates the dispatch table. The dispatch table is indexed by procedure number and contains the addresses of all the XDR filter routines as well as the addresses of the actual procedures.

RPC ПРОГРАМИРАНЕ**:**

ONC RPC- facility developed by the Open Network Computing (ONC) group at Sun Microsystems. The development of RPC applications can be greatly simplified through the use of **rpcgen**, the protocol compiler. rpcgen has its own input language which is used to declare programs, their procedures and the data types for the procedures' parameters and return values.

**VII)** **JAVA REMOTE OBJECT INVOCATION (RMI).** RMI applications often comprise two separate programs, a server and a client. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them*. RMI provides the mechanism by which the server and the client communicate and pass information back and forth*. 1) **Locate remote objects.** Applications can use various mechanisms to obtain references to remote objects. For example, an application can register its remote objects with *RMI's simple naming facility, the RMI registry.* Alternatively, an application can pass and return remote object references as part of other remote invocations.

2) **Communicate with remote objects.** Details of communication between remote objects are handled by RMI. To the programmer, remote communication looks similar to regular Java method invocations. 3) **Load class definitions for objects that are passed around.** Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

THE GENERAL RMI ARCHITECTURE**:** 1) The server must first bind its name to the registry 2) The client lookup the server name in the registry to establish remote references.

3) The Stub serializing the parameters to skeleton, the skeleton invoking the remote method and serializing the result back to the stub.

THE STUB AND SKELETON: 1) A client invokes a remote method, the call is first forwarded to stub. 2) The stub is responsible for sending the remote call over to the server-side skeleton 3) The stub opening a socket to the remote server, marshaling the object parameters and forwarding the data stream to the skeleton. 4) A skeleton contains a method that receives the remote calls, unmarshals the parameters, and invokes the actual remote object implementation.

**VII cont’d 1)** STEPS FOR DEVELOPING AN RMI SYSTEM:

1) Define the remote interface. A remote interface specifies the methods that can be invoked remotely by a client. Clients program to remote interfaces, not to the implementation classes of those interfaces. The design of such interfaces includes the determination of the types of objects that will be used as the parameters and return values for these methods. 2) Develop the remote object by implementing the remote interface. Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces and methods that are available only locally. If any local classes are to be used for parameters or return values of any of these methods, they must be implemented as well. 3) Develop the client program. Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed. 4) Compile the Java source files. 5) Generate the client stubs and server skeletons. 6) Start the RMI registry. 7) Start the remote server objects. 8) Run the client.

Step 1: **Пример:**

import java.rmi.\*;

public interface SampleServer extends Remote **{**

public int sum(int a,int b) throws RemoteException; **}**

Step 2: **Пример:**

import java.rmi.\*; import java.rmi.server.\*;

import java.rmi.registry.\*;

public class SampleServerImpl **extends UnicastRemoteObject** implements SampleServer **{**

SampleServerImpl() throws RemoteException **{** super(); **}**

public int sum(int a,int b) throws RemoteException **{**

return a + b; **}** **}**

The server must bind its name to the registry, the client will look up the server name. Use java.rmi.Naming class to bind the server name to registry. In this example the name call “SAMPLESERVER”.

**VII cont’d 2)** In the main method of your server object, the RMI security manager is created and installed.

public static void main(String args[]) **{**

try **{**

System.setSecurityManager(new RMISecurityManager());

SampleServerImpl Server = new SampleServerImpl();

Naming.rebind(“//localhost/SAMPLE" , Server);

System.out.println("Server waiting…"); **}**

catch (java.net.MalformedURLException me) **{**

System.out.println("Malformed URL”); **}**

catch (RemoteException re) **{**

System.out.println("Remote exception"); **}** **}**

Step 3: **Пример:**

import java.rmi.\*; import java.rmi.server.\*;

public class SampleClient **{**

public static void main(String[] args) **{**

System.setSecurityManager(new RMISecurityManager());

try **{**

System.out.println("Security Manager loaded");

String url = "//localhost/SAMPLE";

SampleServer remoteObject = (SampleServer) Naming.lookup(url);

System.out.println("Got remote object");

System.out.println("1+2="+remoteObject.sum(1,2)); **}**

catch (RemoteException exc) **{**

System.out.println("Error in lookup"); **}**

catch (java.net.MalformedURLException exc) **{**

System.out.println("Malformed URL"); **}**

catch (java.rmi.NotBoundException exc) **{**

System.out.println("NotBound"); **}** **}** **}**

Step 6: 1) The RMI applications need install to Registry. And the Registry must start manual by call rmiregisty. 2) The rmiregistry us uses port 1099 by default.

**VII cont’d 3)** JAVA POLICY FILE: 1) In Java 2, the java application must first obtain information regarding its privileges. It can obtain the security policy through a policy file. In above example, we allow Java code to have all permissions, the contains of the policy file policy.all is:

grant **{** permission java.security.AllPermission; **};**

2) Now, we given an example for assigning resource permissions:

grant {

permission java.io.filePermission “/tmp/\*”, “read”, “write”;

permission java.net.SocketPermission “host.on.com:999”, ”connect”;

permission java.net.SocketPermission “\*:1024-65535”, ”connect,request”;

permission java.net.SocketPermission “\*:80”, ”connect”; **};**

COMMENT FOR THE JAVA POLICY FILE: 1) allow the Java code to read/write any files only under the /tmp directory, includes any subdirectories 2) allow all java classes to establish a network connection with the host “somehost.somedomain.com” on port 999 3) allows classes to connection to or accept connections on unprivileged ports greater than 1024 , on any host 4) allows all classes to connect to the HTTP port 80 on any host.

**VIII) HTTP.** HTTP е протокол, който е бил разработен за търсене и публикуване на хипертекстови страници. Впоследствие получава възможност за обслужване на хипермедиаинформационни системи (процесинг на разнобразн типове от данни). **ОСНОВНИ СВОЙСТВА:** 1) Stateless protocol; 2) TCP клиент-сървер протокол: Клиентът (браузер, бот и т.н.) се нарича общо потребителски агент**;** Между агента и сървера може да имa междинни звена – проксита и тунели. 3) Request-response протокол – на всяка заявка на агента сърверът е длъжен да даде съответния отговор. 4) При версиите 0.9 и 1.0 за всяка двойка request-response се създава отделна конекция. При версията 1.1 е възможна постоянна конекция, през която да минат няколко request-respnse-a. 5) Псевдо-анонимни клиенти. През протокола се изпраща само IP адреса на агента. Информацията за потребителя следва да се включи в тялото на request-а.- Стандартно се използва порт 80. Връзки между клиента и сървъра: директна, през прокси или през тунел;

**ФОРМАТ НА REQUEST:** request-line; [header]…; blank line; [body]

Където формата на **request-line** е **method request-URL HTTP-version** Методът може да е един от следните: HEAD, GET, POST, PUT, DELETE, TRACE или OPTIONS. **REQUEST МЕТОДИ:** GET: Requests a representation of the specified resource. By far the most common method used on the Web today. Should not be used for operations that cause side-effects (using it for actions in web applications is a common misuse). HEAD: Asks for the response identical to the one that would correspond to a GET request, but without the response body. This is useful for retrieving meta-information written in response headers, without having to transport the entire content. POST: Submits data to be processed (e.g. from an HTML form) to the identified resource. The data is included in the body of the request.This may result in the creation of a new resource or the updates of existing resources or both.

**XIII cont’d 1)** PUT: Uploads a representation of the specified

resource. DELETE: Deletes the specified resource. TRACE: Echoes back the received request, so that a client can see what intermediate servers are adding or changing in the request. OPTIONS: Returns the HTTP methods that the server supports. This can be used to check the functionality of a web server. НАДЕЖДНИ МЕТОДИ: Методите HEAD, GET, OPTIONS, and TRACE са надеждни, защото нямат страничен ефект, т.е. не изменят състоянието на сървера.

**RESPONSE**: *status-line; [header]…; [blank line]; [Body]*

Където форматът на **status-line** е **HTTP-version response-code responsephrase Примери:**

200 Success**;** 201 OK, request succeeded**;** 400 Client error**;** 401 Bad request**;** 402 Unauthorized request**;** requires user authentication**;** 403 Forbidden for unspecified reason**;** 404 Not found**;** 500 Internal server error.

**HEADERS (ПРИМЕРИ):** Accept: text/plain**;** Accept-Charset: iso-8859-5**;** Accept-Encoding: compress, gzip**;** Content-Length: 348

**URL:** Синтаксис:

scheme name : hierarchical part [ ? query ][ # fragment]

hierarchical part://[user info@]hostname[:port][path]

query:*key1*=*value1*[&k*ey2*=*value*2]…

fragment: допълнителна информация

URL (пример):

**Request-respose (пример):**

**GET / http/1.0**

HTTP/1.1 200 OK

Date: Fri, 07 Oct 2005 15:17:19 GMT

Server: Apache/2.0.48 (Unix) mod\_ssl/2.0.48 OpenSSL/0.9.7c

DAV/2 PHP/4.3.4

Last-Modified: Thu, 08 Sep 2005 06:27:29 GMT

ETag: "450923-11df-c15d5640"

Accept-Ranges: bytes

Content-Length: 4575

Connection: close

Content-Type: text/html; charset=ISO-8859-1

**XIII cont’d 2) Cookies:** A cookie is a small piece of data containing some user-specific information. Cookies may be created/stored the following ways: 1) Generated by client to reside on client site. 2) Generated by server to reside on client site. 3) Generated by server to reside on server site.

**MIME:** (Multiple Internet Mail Extensions) A standard used to encode binary data as printable ASCII text for transmission across the Internet.Format of a MIME type:**type/subtyp**

**HTTPS:** HTTP протокол, предаван върху SSL (secure Socket layer). Стандартно използва 443 порт. Агентът и сърверът трябва да поддържат SSL. На сърверът трябва да бъде инсталиран сертификат. Принцип на действие: Инициира се заявка за скрита връзка от агента(клиента). Сърверът изпраща своят сертификат. Агентът проверява сертификата. Агентът изпраща своя сертификат на сървера с инфо, криптирано с публичен ключ от сертиификата на сървера. На базата на това инфо, сърверът и агентът се договарят за параметрите на скритата връзка.

**IX) HTML + CGI:** HTML PROGRAMMING:

Client side processing: Java applets;-JavaScript;

Server side processing: CGI;-PHP;

Java Tools – servlets, JSP, JEE

Data base connections;

CLIENT SIDE PROCESSING Изпълн. на програмир. Модули се извършва в средата на потребителския агент. Java аплетите могат да вървят в Web browser използв. Java Virtual Machine (JVM), или в Sun's AppletViewer, самостоят.средство за тестване на аплети.

Активиране на аплет – HTML tag

**<APPLET** CODE = appletFile WIDTH = pixels HEIGHT = pixels [ALIGN = alignment] [VSPACE = pixels] [HSPACE = pixels]**> [<PARAM** NAME = appletParameter1 VALUE = value**>] [<PARAM** NAME = appletParameter2 VALUE = value >**] … </APPLET>**

ЖИЗНЕН ЦИКЪЛ НА АПЛЕТА:

import java.awt.\*; import java.applet.Applet;

public class AppletStructure extends Applet **{**

public void init() **{** System.out.println("initializing"); **}**

public void start() **{** System.out.println("starting"); **}**

public void paint(Graphics g) **{**

System.out.println("painting");

g.drawString("Hello World!", 30, 30); **}**

public void stop() **{** System.out.println("stopping"); **}** **}**

**init () method:** Живота на аплета започва, когато за пръв път е зареден в брауз и е извикам Init метод. init() method се извикв. само веднъж. init() се вика за да се прочете PARAM tag в HTML файла. **start () method:** Може да се вика мн пъти, когато аплета тр да се стартира или рест. **stop () method:** също може да вика мн пъти. **destroy() method:** вика се веднъж в живота на аплета, когато the browser тр да се изключи.

JAVASCRIPT: JavaScript was designed to add interactivity to HTML pages. JavaScript is a scripting language. JavaScript is an interpreted language (means that scripts execute without preliminary compilation). JavaScript използва HTML DOM.

**IX cont’d 1)** The HTML Document Object Model (HTML DOM) defines a standard way for accessing and manipulating HTML documents. The DOM presents an HTML document as a treestructure (a node tree), with elements, attributes, and text**.** According to the DOM, everything in an HTML document is a node.

SERVER SIDE PROCESSING: Програмните модули се изпълняват на сървъра и генерират HTML съд., което сърверът изпраща към клиента.

COMMON GATEWAY INTERFACE (CGI): HTTP request-а може да реферира два вида ресурси: 1) Файл от файл с-ма на сървера. Тогава сърверът връща съдържанието на този файл. 2) Изпълним модул (програма, командна процедура, скрипт,...). Тогава сърверът приема инфо, която предава на изпълнимия модул и връща неговият изход. CGI дефинира стандартите, които се следват във втория случай - как сърверът предава на изпълнимият модул изпратената от агента инфо и как сърверът може да добави допълнителна инфо към изпратения изход (нпр. чрез header-ите). CGI МЕТОДИ: **GET**(request парам вURL) инфото от агента се включва в URL-а като параметри. CGI модулът намира тази информация в променлива на обкъжението. **POST**(request param in body) – инфото се изпраща в тялото на request-а, а CGI модулът се чете от станд. си вход.

CGI РЕАЛИЗАЦИЯ: 1) CGI модулът се активира чрез създаване на нов процес. 2) Процесът получава изпратената от агента инфо съгласно на дефинираният метод. 3) Процесът произвежда станд. изход, който се изпраща обратно на агента. 4) Процесът завършва своята работа. ОПТИМИЗАЦИЯ НА РЕАЛИЗАЦИЯТА: Fast CGI –сърверът използва постоянен процес, където се изпълняват CGI модулите (частично реализиран вApache – **mod\_fastcgi**)**;** Сърверни модули – CGI процесорите се включват в тялото на Web сървера**;** Резидентни процеси (ако OS ги поддържа).

**IX cont’d 2)** ФОРМИ: Начин за пред. на данни от агента (клиента) към сървера.

**<FORM ACTION=*cgi*-*модул* METOD={GET |POST}>**

***Html оператори и form контроли***

**</FORM>**

FORM контроли: Всяка контрола има име и получава Евент. ст-ст. Формата генерира URL от вида

**http://*cgi*-*модул*?*и1=с*1&*и2=с2*&...>**

където *иj* и *сj* са името и стойността на j-тата контрола.

ТЕКСТОВИ КОНТРОЛИ: **Пример:**

**Enter your name: <INPUT NAME=name TYPE=“TEXT” SIZE=20 MAXLENGTH=40> and password: <INPUT NAME=passwd TYPE=“PASSWORD” SIZE=8 MAXLENGTH=8><BR> <INPUT TYPE=“HIDDEN” NAME=*име* VALUE=*стойност*> Please enter your address:<BR> <TEXTAREA NAME=address ROWS=5 COLS=50> </TEXTAREA>**

Списъци от опции: **<INPUT TYPE=“CHECKBOX” NAME = *име* VALUE = *стойност* [CHECKED]>** Изпълнителни контоли **<INPUT TYPE=“SUBMIT” NAME=име VALUE=текст> <INPUT TYPE=“RESET” VALUE=текст>**

**X) PHP.** Включване на PHP скрипт в HTML файл: <?php

php оператори…?> Изходът от скрипта се вклчва в HTML-а. <! DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/TR/REC-html40/loose.dtd"> <HTML> <HEAD> <TITLE>Hello</TITLE></HEAD><BODY>Hello world!<BR>

<?php print("PHP salutes you");?></BODY></HTML>, е еквивалентно на - **Hello world!\nPHP salutes you**. **Не смесвайте PHP и HTML оператори в PHP**

**Скрипт**: <?php print("PHP salutes you"); <BR> echo "Next line" ?> **Свободна типизация** – типът на данните се

определя от контекста, в който тя се появява: <?php

print("PHP data types<BR>");

$var = 1 + 2;

print("VAR is $var. <BR>");

$var ="one " ;

print("VAR is $var. <BR>");?>

Резултат: PHP data types\nVAR is 3.\nVAR is one .

**Типове скаларни данни:** цели числа, плаваща точка,

булеви типове (TRUE и FALSE), символни низове. **Масивите са асоциативни.($e [1] = 10;$e [100] = 1000;**

<?php

$note["Geometry"] = 4;

$note["OS"] = 6;

$note["Num methods"] = 3;

reset($note);

$value = current($note);

$key = key($note);

print ("$key = $value <br>");

while ($value=next($note)){

$key = key($note);

print ("$key = $value <br>");}

?>

Резултат: Geometry = 4

OS = 6

Num methods = 3.

**X cont’d 1) Операции и Управляващи структури – аналогични на C.** If ..else.. elseif/else if; while; do-while;

For; foreach (array\_expression as $value); statement; break, continue; switch; return; require ,include;

**Обектен модел** -

**class** SimpleClass [extends ParentClass] {

**// дефиниция на член**

public $var = 'стойност по подразбиране';

**// дефиниция на конструктор**

void \_\_construct (аргументи ) { }

**// дефиниция на метод**

public function displayVar() {

echo $this->var; }

}

**Accessing the values of the form controls** –

<form action="foo.php" method="post">

Name: <input type="text" name="username" /><br />

Email: <input type="text" name="email" /><br />

<input type="submit" name="submit" value="Submit me!" />

</form>

<?php // Available since PHP 4.1.0

echo $\_POST['username'];-достъп до дата след form “post”

echo $\_REQUEST['username'];?>-след form “post” или“get”

**Автентичност**: Set the configuration to use this password file - **базисен метод** - AuthType Basic;

AuthName "By Invitation Only"; AuthUserFile/usr/local/apache/passwd/password;

Require user rbowen sungo.

**Защитен**-(digest) metod - AuthType Digest

AuthName "Private";

AuthDigestFile/usr/local/apache/passwd/digest;

Require user drbacchus dorfl.

**И при двата метода сърверът изпраща header 401 Authentication Required при обръщение към защитен ресурс.**

**X cont’d 2) PHP авторизация:**

**<?php**

**if (!isset($\_SERVER['PHP\_AUTH\_USER'])) {**

**header('WWW-Authenticate: Basic realm="My Realm"');**

**header('HTTP/1.0 401 Unauthorized');**

**echo 'Text to send if user hits Cancel button';**

**exit;**

**} else {**

**echo "<p>Hello {$\_SERVER['PHP\_AUTH\_USER']}.</p>";**

**echo "<p>You entered {$\_SERVER['PHP\_AUTH\_PW']} as your password.</p>";**

**}**

**?>**

**XI) JAVA Web Components:**

**Servlets** are Java programming language classes that

dynamically process requests and construct responses.

**JSP pages** са текстово-базирани документи, които се изпълнява като сървлети, но позволяват по-естествен подход за създаване на статично съдържание.

**Web container** съдържа: a basic web server;a request/response translator;a runtime environment for the web components;supports specific objects and methods;

**Servlets** - life cycle - 1.If an instance of the servlet does not exist, the Web container: Loads the servlet class, Creates an instance of the servlet class, Initializes the servlet instance by calling the init method. 2. When the request is received it invokes the service method. 3.service calls doMethod according the Method specified in the HTTP request and passes to it a request and response object. 4. When the servlet is removed or reloaded invokes destroy method.

**The servlet is an Java class, which extends the**

**base class HttpServlet.** import java.io.\*; import javax.servlet.\*; import javax.servlet.http.\*; public class <ServletName> extends HttpServlet {//servlet methods}

**XI cont’d 1) Initializing a Servlet** -After the Web container loads and instantiates the servlet class and before it delivers requests from clients, the Web container initializes the servlet. You can customize this process to allow the servlet to read persistent configuration data, initialize resources, and perform any other one-time activities by overriding the init method of the Servlet interface. А servlet that cannot complete its initialization process should throw **UnavailableException.**

public class CatalogServlet extends HttpServlet {

private BookDB bookDB; public void init() throws ServletException { bookDB = OpenDB(“Book DB”);

if (bookDB == null) throw new UnavailableException("Couldn't get database.");}}

The service provided by a servlet is implemented in the

service method of a **GenericServlet**. It invokes the

**doMethod** methods (**where Method can take the value Get, Delete, Options, Post, Put, Trace**).

public void doGet (HttpServletRequest request,

HttpServletResponse response) throws ServletException, IOException {}

Getting Information from **Requests - Parameters**, which are typically used to convey information between clients and

servlets: String bookId = request.getParameter("Add");

if (bookId != null) {}

**Constructing Responses: -** Retrieve an output stream to use to send data to the client. To send character data, use the PrintWriter object returned by the response's getWriter method; - Indicate the content type (for example, text/html) being returned by the response with the setContentType (String) method. - Indicate whether to buffer output with the

setBufferSize(int) method. response.setContentType("text/html"); response.setBufferSize(8192); PrintWriter out = response.getWriter(); out.println("<html>" +"<head><title> + "TitleBookDescription"+</title></head>");

**XI cont’d 2) JSP** - JSP page is a text document that contains two types of text: **static data**, which can be expressed in any text-based format (such as HTML, SVG, WML, and XML), and **JSP elements**, which construct **dynamic content**.

**JSP Expression** <%= expression %> is evaluated and placed in output. **JSP Scriptlet** <% code %> Code is inserted in service method. **SP Comment** <%-- comment --%> - ignored when JSP page is translated into servlet. **SP include Directive** <%@ include file="url" %> a file on the local system to be included when the JSP page is translated into a servlet.

**Predefined objects: -** request, the HttpServletRequest; - response, the HttpServletResponse; - session, the HttpSession associated with the request (if any); - out, the PrintWriter (a buffered version of type JspWriter) used to send output to the client.

Your hostname: <%= request.getRemoteHost() %>

Your name: <%= request.getParameter(“Name”) %>

<html><head><title>Sample Application JSP Page</title>

</head><body bgcolor=white>

<CENTER><img src="images/tomcat.gif">

<%= new String("<BR>Tomcat salutes you!<BR>") %>

</CENTER><%= "The request is sent from " +request.getRemoteHost() %>

<%

String queryData = request.getQueryString();

if (queryData == null)

out.println("<BR> No parameters were sent!");

else

out.println("<BR>Parameters are:" + queryData);

%> </body></html>

**Access to CGI variables:**

"AUTH\_TYPE", request.getAuthType() ,

"CONTENT\_LENGTH",String.valueOf(request.getContentLength())

"CONTENT\_TYPE", request.getContentType()

"DOCUMENT\_ROOT", getServletContext().getRealPath("/")

"PATH\_INFO", request.getPathInfo()

"PATH\_TRANSLATED", request.getPathTranslated()

"QUERY\_STRING", request.getQueryString()

"REMOTE\_ADDR", request.getRemoteAddr()

"REMOTE\_HOST", request.getRemoteHost()

"REMOTE\_USER", request.getRemoteUser()

"REQUEST\_METHOD", request.getMethod()

"SCRIPT\_NAME", request.getServletPath()

"SERVER\_NAME", request.getServerName()

"SERVER\_PORT", String.valueOf(request.getServerPort())

"SERVER\_PROTOCOL", request.getProtocol()

"SERVER\_SOFTWARE",getServletContext().getServerInfo()

**XII) Web архитектури.** Model View Controller: (MVC)

Controller: Взима данните от модела или извиква доп. методи върху Model, предварително обработва данните, и чак след това ги дава на View.

Model: ядрото на app-а, дефинирано от областта, за която се разработва; обикновено това са данните от реалния свят, които сме моделирали и над които искаме да работим. Вика се от Controller. БД работят тук.

View: Tази част от изходния код на приложението, отговорна за показването на данните от модела. Mоже да се състои от PHP шаблонни класове, JSP страници, ASP страници…

Ruby on Rails: Framework за Ruby, използва MVC. Може да създава части от разработката автоматично. Със WEBrick (прост Ruby сървър) и Rake (система за build-ване) създава среда за разработване.

PHP MVC frameworks: Akelos PHP Framework (порт на RoR за PHP4/5)**;** Barebonesmvc (PHP5, еднофайлов)**;** CodeIgniter**;** ash.MVC**;** FUSE, CakePHP, Zend Framework, Symfony Framework (последните 4 взаимстват от RoR).

Многослойна архитектура: Тier 0: Browser (PHP: JavaScript**;** Java: JavaScript, Applets) ↔ Presentation (PHP + HTML**;** Java: JSP, Servlets) ↔ Business Logic (PHP classes**;** Java: Servlets, Java Classes) ↔ DB Engine (PHP: MySQL**;** Java: Oracle, etc)

Enterprise Java Beans: (EJB) По-добър от RMI, защото обхваща: сигурност, търсене, транзакции, запазване на данните. Управляват се от J2EE сървър като негови компоненти. Сървъра управлява ресурсите им.

Java Enterprise Beans: Session bean: Stateful – съдържа данни за някакво състояние; Stateless – позволяват конкурентен достъп. Entity Beans – запазване на данни. Message Driven Beans – управ. асинхр. JMS съобщения**.**

**XII cont’d)** Java Enterprise Beans: Интерфейса декларира бизнес и life-cycle методи за bean-a. **Пример:**

@Remote // Бизнес интерфейс на bean-а

public interface InterfaceName {...}

@Remote(InterfaceName.class)

public class BeanName implements InterfaceName {...}

Отдалечен (remote) достъп в ЕЕ5: Клиента работи в отделна (може и същата) вирт. машина (JVM) от bean-а, който достъпва. Може да е web компонент, app client, или друг ent. bean. За клиента мястото на bean-a е невидим.

Локален достъп в ЕЕ5: Задължително е в същата JVM като bean-a. Клиента е web component или друг ent. bean. За локалния мястото на bean-а е видимо. За локални bean интерфейси може, но не е задължително да се анотират с @Local (взема се по подразбиране).

Избор между локален и отдалечен достъп: Близко свърз. bean-ове са кандидати за локален. Ако се достъпва от app client-и или се дистрибутира трябва да е отдалечен.

Stateless Bean: **Пример:**

package com.sun.tutorial.javaee.ejb;

import java.math.BigDecimal;

import javax.ejb.\*;

@Remote

public interface Converter **{**

public BigDecimal dollarToYen(BigDecimal dollars);

public BigDecimal yenToEuro(BigDecimal yen); **}**

@Stateless

public class ConverterBean implements Converter **{**

private BigDecimal yenRate = new BigDecimal("115.31");

private BigDecimal euroRate = new BigDecimal("0.007");

public BigDecimal dollarToYen(BigDecimal dollars) **{**

BigDecimal result = dollars.multiply(yenRate);

return result.setScale(2, BigDecimal.ROUND\_UP); **}**

public BigDecimal yenToEuro(BigDecimal yen) **{**

BigDecimal result = yen.multiply(euroRate);

return result.setScale(2, BigDecimal.ROUND\_UP); **}** **}**

**XIII) Бази Данни.** Защо БД: Независимост от физ. представяне**;** Управление на данните и метаданните**;** Управление на достъпа**;** Безопасност и сигурност на данните**;** Архивиране (on fly) и възстановяване**.**

Релационни БД: Една релационна БД се състои от двумерни масиви (наричани още таблици или релации). Всяка колона от таблица има свое име. БД е в **първа нормална форма**, ако стойностите в редовете на таблиците са **скалари.**

Ключове: Един или повече атрибута със специално предназначение в таблицата на релацията. 4 вида: кандидат ключ**;** първичен ключ – идентифицира по уникален начин всеки запис**;** външен ключ – необходим, когато имаме отношение между две таблици**;** алтернат. ключ**.**

Връзки между таблиците: (Relation) Е зависимост, съществуваща между две таблици, когато записи от първата таблица могат да се свържат по някакъв начин със записи от втората таблица. 1:1, 1:N, M:N**.**

Езици за управление: Structured Query Language – SQL е най-популярният език за манипулация с релационни БД. Заявките, които се отправят към релационната БД, и извлечените в резултат подтаблици се изразяват със средствата на релационната алгебра (рел. смятане): Селекция (SELECT)**;** Обединение (UNION)**;** Сечение (INTERSECT)**;** Разлика (EXCEPT, MINUS)**;** Дек. произв. (CROSS JOIN)**;** Селекция (SELECT)**;** Проекция – селекция махаща повтарящи се записи (GROUP BY, DISTINCT)**;** Съединение (LEFT/RIGHT JOIN)**.**

SQL Оператори: Създаване (CREATE)**;** Вмъкване (INSERT)**;** Изтриване(DELETE)**;** Актуализиране (UPDATE) **Пример:**

INSERT INTO My\_table (col1, col2, col3) VALUES ('test', 'N', NULL);

UPDATE My\_table SET col1 = 'updated value' WHERE col2 = 'N';

DELETE FROM My\_table WHERE col2 = 'N';

SELECT \* FROM T WHERE C1 = 1;

**XIV) Връзка към бази от данни.** Директна връзка: Web application ↔ Database server. PHP + MySQL: mysql\_connect($host, $user, $password) – свързване; mysql\_select\_db($database) – селектиране на БД; $result = mysql\_query($SQL\_expresion) – изп. заявка; **Пример:**

**<?php** $con = mysql\_connect("localhost", "root");

$res = mysql\_select\_db("student");

mysql\_query("SELECT \* FROM notes where fn=1001");

echo mysql\_affected\_rows(); **?>**

mysql\_affected\_rows() връща всички селект. записи**;** mysql\_fetch\_row($res) – следващия запис в масив**;** mysql\_fetch\_object($res) – next запис като обект**;** mysql\_fetch\_array($res) – следващия запис като асоциативен масив**;** **Пример:**

**………………………………………………………………………………………………………………………………………………**

$con = mysql\_connect("localhost", "root", “t00r”);

$res = mysql\_select\_db("student");

$result = mysql\_query("<горния пример за SQL>");

while ($row=mysql\_fetch\_array($result)) **{**

echo $row["fn"],":", $row["course\_code"], ":", $row["note"], "<BR>"; **}**

Недостатъци на директ. връзка: Ф-иите са разл. за всяка БД: mysql\_query() ↔ ibase\_query() ↔ pg\_exec(). Кодът **не** може да бъде преизползван за друг сценарий**!**

Връзка с Connector: Web app. ↔ Connector ↔ DB server (известни connector-и са PDO, ODBC, JDBC). PDO (PHP Data Objects): Връзка независима от БД с архитектура на 2 нива. Горното ниво е интерфейс за връзката без знач. от вида на БД-то. Долното – имплементация на драйвър специфичен за БД-то. За връзка се задава DSN (data source name) и параметри (user и pass не са задълж.) $dbh = new PDO("mysql:host=localhost;dbname=test","user","pass")**;** За селект. заявки се ползва (връща обект PDOStatement) $sth = $dbh->query($SQL\_expresion)**;** За останалите заявки: $count = $dbh->exec("some SQL statement")**;**

**XIV cont’d 1)** Резулт. на PDO се извличат по 2 начина. $row = $sth->fetch(PDO::FETCH\_NUM) като масив (row[0]) и $row = $sth->fetch(PDO::FETCH\_ASSOC) като асоц. масив (по име на колона row[“name”])**.**

Java SQL interface: Tier (ниво) 1: Java Server Pages – JSP (презентация) ↔ Tier 2: Java Classes (бизнес логика) ↔ Tier 3: DB engine (през SQL с JDBC)**.** JDBC – RDBMS Java Interface: Регистрираме RDBMS драйвъра и викаме getConnection(). **Пример:**

import java.sql.\*;

**………………………………………………………………………………………………………………………………………………**

Connection conn = null;

try **{**

String userName = "user";

String password = "pass";

String url = "jdbc:mysql://localhost/test";

Class.forName("com.mysql.jdbc.Driver").newInstance();

conn = DriverManager.getConnection (url, userName,

password);

System.out.println("DB connection established"); **}**

catch (Exception e) **{**

System.err.println("Cannot connect to DB server"); **}**

За заявки без резултат вземаме Statement обект от Connection обект. С executeUpdate() изпълняваме модифицираща заявка. **Пример:**

**………………………………………………………………………………………………………………………………………………**

Statement s = conn.createStatement();

int count;

s.executeUpdate("DROP TABLE IF EXISTS animal");

s.executeUpdate("CREATE TABLE animal (id INT UNSIGNED NOT NULL AUTO\_INCREMENT, PRIMARY KEY(id), name CHAR(40), category CHAR(40))");

count = s.executeUpdate("INSERT INTO animal (name, category) VALUES ('snake', 'reptile'), ('frog', 'amphibian')");

s.close();

System.out.println(count + " rows were inserted");

**XIV cont’d 2)** Заявките с резултат (SELECT) връщат ResultSet, който е Iterable. Изпълняват се чрез executeQuery(). Достъпваме кол. от row по getXXX() (XXX е името ѝ). **Пример:**

**………………………………………………………………………………………………………………………………………………**

Statement s = conn.createStatement();

s.executeQuery("SELECT id, name, category FROM animal");

ResultSet rs = s.getResultSet();

int count;

for(count = 0; rs.next(); count++) **{**

int idVal = rs.getInt("id");

String nameVal = rs.getString("name");

String catVal = rs.getString("category");

System.out.println("id="+ idVal + ",name="+ nameVal + ",category="+ catVal); **}**

rs.close();

s.close();

System.out.println(count + " rows were retrieved");

Обектен подход за достъп до БД: Запазване (Persistance) за данни след живота на програмата (иначе се губят), на HDD или в БД. В обектните системи представяме Entities като обекти (за да се преизползват) и БД ги запазва**.**

Обектен vs. Релационен модел: Обектите не са скалари**;** Данните на обекта се разпределят в една или повече инстанции**;** Проблема за granularity възниква като съпоставянето на класове към таблици не съвпада**.**

Java Persistance API (JPA): Управлява релационни данни и има 3 области: API в javax.persistence пакета**;** Java Persistence Query Language**;** object/relational мета-данни. Tier 1: JSP (презентация) ↔ Tier 2: Java Classes (бизнес логика) ↔ Persistance Layer ↔ Tier 3: DB engine (през SQL с JDBC). Entity: Представя row от БД като обект за лесна интеграция, всяка инстанция е отделен row**.**

**XIV cont’d 3)** Entitiy: **Пример:**

import java.io.Serializable;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity // java анотация

@Table(name=“EMPLOYEE TABLE")

public class Employee **{**

@Id

private int id;

private String name;

public Employee(int id) **{** this.id = id; **}**

public int getId() **{** return id; **}**

public void setId(int id) **{** this.id = id; **}**

public String getName() **{** return name; **}**

public void setName(String name) **{** this.name = name; **}** **}**

Entity Manager: За да се управляват инстанциите на Entity трябва явно да се декларират под контрола на Entiti Manager от JPA. След това сме сигурни че всички промени се синхронизират с БД. След като контрола прикючи Entity става нормален обект. **Пример:**

**………………………………………………………………………………………………………………………………………………**

EntityManager em;

// създаваме нова Entity инстанция (↔ на row в БД)

Employee person = new Employee(10);

person.setName("Miller");

// декларираме контрол от Entity Manager-a

em.persist(person);

**………………………………………………………………………………………………………………………………………………**

person = em.find(Employee.class, 10);

// абонираме Entity за премахване

if(person != null) **{** em.remove(person); **}**